I hereby certify that this correspondence is being deposited with the United States Postal Service "Express Mail Post Office to Addressee" service on the date indicated below and is addressed to Box Patent Application, Assistant Commissioner for Patents, Washington, D.C. 20231.

"Express Mail" label number: **EF414677999US**

Date: **May 24, 2001**

By: [illegible]
John A. Smart

Docket No. **LS/0026.00**

PATENT APPLICATION

ORDER SCHEDULING SYSTEM AND METHODOLOGY

Inventor: JOHN RODRIGUEZ, a citizen of The United States residing in Capitola, CA.

Assignee: LightSurf Technologies, Inc.

John A. Smart
Reg. No. 34,929

ORDER SCHEDULING SYSTEM AND METHODOLOGY

COPYRIGHT NOTICE

BACKGROUND OF THE INVENTION

1. Field of the Invention

The present invention relates to the field of order fulfillment and, more particularly, to system and methodology for efficiently managing order fulfillment.

2. Description of the Background Art

Despite advances afforded by e-commerce, a fundamental problem still exists today in terms of how to efficiently fulfill an order that has been placed by a customer. This problem typically faces those who take customer orders, that is, the "middlemen" (which is used herein to broadly refer to retailers, distributors, or the like). Often, a middleman will have to send a customer order to a "fulfiller," that is, an organization that will fulfill the order by actually shipping ordered goods back to the customer. The considerations involved in choosing a particular fulfiller are numerous, but typically a middleman chooses a fulfiller with the primary goal of minimizing costs, thus maximizing profits. Different cost-related constraints may come into play, when striving to maximize profit. For example, in addition to the cost incurred as a result of the price charged by a given fulfiller, other cost considerations include what shipping charges are incurred when shipping from a given fulfiller. One fulfiller may have a better price, but that price advantage may be negated by unfavorable shipping charges. Further, cost is not the only factor to consider. Instead, at least some consideration must be given to the ability of a particular fulfiller to timely fulfill an order. There is no point in choosing a fulfiller who offers the best price, if that fulfiller lacks sufficient inventory to successfully fulfill the order in a reasonable period of time.

Often an order cannot be completely fulfilled, or supplied, by a single fulfiller; therefore, the fulfillment of such an order is spread across multiple fulfillers. The determination of the distribution of sub-orders, or product orders within an order, to multiple fulfillers is the *scheduling* of order shipments for that order. Order shipment scheduling attempts to optimize the fulfiller distribution to minimize the shipping costs to either the customer or to the middleman processing the order. Minimizing the shipping costs almost invariably indicates minimizing the number of fulfillers satisfying an order. Optimized scheduling also minimizes delivery time, and satisfies any arbitrary business logic, such as favoring specified fulfillers (when more than one fulfiller can deliver the same order item in an order). The middleman needs to efficiently optimize the order shipment(s) scheduling according to whatever constraints he or she uses as criteria for assigning order items to fulfillers.

Previous attempts to automate (by computer programming) the optimization of order shipment scheduling have not been satisfactory. Current methods involve considerable time for programming development/updating, and require costly compute time. The prevailing approach uses the simplex method for solving the linear programs comprising multiple simultaneous linear equations; see, e.g., Anderson, David Ray, *An Introduction to Management Science: Quantitative Approaches to Decision Making*, Seventh Edition, Chapter 5 (particularly at pp. 190-192), West Publishing Company, 1994, the disclosure of which is hereby incorporated by reference. This approach develops a linear programming model comprising a set of linear equations: each linear equation describes a constraint (e.g., proximity of shipper-to-shipping recipient) to be applied to all potential fulfillers.

A linear equation solves for a single variable, and takes the form: $ax + b = 0$, where x is the unknown variable, and both a and b are constant numerical values. In linear equations, the variable, x, always has its exponential value set to 1; exponential or logarithmic variable types are not employed as operands in linear equations. Each linear equation can be graphed as a straight line in a two-dimensional XY-coordinate plane. The coefficient for the variable (the constant numerical value of a, in the generic form) determines the slope of the straight line for that equation. This equation is processed for every fulfiller considered. If the scheduling method implements multiple constraints, then a separate linear equation is processed against each constraint simultaneously. Multiple linear equations can be mapped onto a two-dimensional graph. The set of possible solutions (that minimizes for these constraints) is bound by the area beneath the intersections of the straight line on the graph.

Fig. 1 is an XY two-dimensional coordinate graph showing the slopes for three separate linear equations representing three constraints in a problem for scheduling types of personal computers (e.g., DeskPro™): warehouse capacity, display units, and assembly time. Fig. 1 includes the slope 100 for the equation constraining warehouse capacity, the slope 110 for the equation constraining display units, the slope 120 for the equation constraining assembly time, the area-bounding intersection 130 of the X and Y axes at value (0,0), the area-bounding intersection 140 of the warehouse capacity slope 100 and the X-axis, the area-bounding intersection 150 of the assembly time slope 120 and the warehouse capacity slope 100, the area-bounding intersection 160 of the display units slope 110 and the assembly time slope 120, and the area-bounding intersection 170 of the Y-axis and the display units slope 110. The area bound by the intersections in Fig. 1, 130, 140, 150, 160, and 170, contains the *set* of feasible solutions for this problem. A discrete solution for any variable (constraint) can be determined by holding all the other variables' values at a constant (within the feasible set).

Current systems using linear programming with multiple simultaneous equations leave much to be desired. A well-known problem with linear solutions is that the simplex method requires intensive iteration. Computerized solutions for multiple simultaneous equations are therefore time-consuming. Another deficiency with linear programming is the programmatic difficulty in setting-up the equations. Because the constraints can be described in linear equations with inequalities, it is challenging to program a general solution that applies to every scenario. Full appreciation of all the factors defining the constraints cannot be completely known a priori. For example, if the program implements a policy towards minimizing shipping costs, the program would need to know all of the distances between the location of the middleman or the customer and the location of every fulfiller: that information would have to be put into a database, and extracted-out and placed into a coefficient for each iteration of each equation. The approach has a degree of fuzziness that stems from the implicit effects of other incidental variables, such as *slack time* (which is the consequence of having determined the best solution, there is always a remainder left over).

The simplex method of solving linear equations is not the only method; however, the other methods are even more difficult to implement. Because of the ever-increasing demands of the marketplace (and e-commerce marketplace) for timely, cost-effective fulfillment of customer orders, much interest exists in finding a solution to these problems.

GLOSSARY

bit vector: A one-dimensional array of bit (e.g., 1 and 0) values, which is useful for specifying a sequence of Boolean (true/false) values.

fulfiller: Fulfiller, as used herein, broadly refers to any entity capable of fulfilling an order or portions (i.e., order items) thereof, which may include a distributor, supplier, vendor, manufacturer, service bureau, or the like. Typically, the fulfiller receives orders from a middleman (e.g., retailer, or the like). The fulfiller may fulfill an order by shipping directly to the end customer, or by shipping to the middleman (who in turn ships to the end customer).

HTTP: Short for *HyperText Transfer Protocol*, the underlying protocol used by the World Wide Web. HTTP defines how messages are formatted and transmitted, and what actions Web servers and browsers should take in response to various commands. For example, when a user enters a URL in his or her browser, this actually sends an HTTP command to the Web server directing it to fetch and transmit the requested Web page. Further description of HTTP is available in *RFC 2616: Hypertext Transfer Protocol -- HTTP/1.1*, the disclosure of which is hereby incorporated by reference. *RFC 2616* is available from the World Wide Web Consortium (W3), and is currently available via the Internet at *http://www.w3.org/Protocols/.*

Java: A general purpose programming language developed by Sun Microsystems. Java is an object-oriented language similar to C++, but simplified to eliminate language features that cause common programming errors. Java source code files (files with a .java extension) are compiled into a format called bytecode (files with a .class extension), which can then be executed by a Java interpreter. Compiled Java code can run on most computers because Java interpreters and runtime environments, known as Java Virtual Machines (JVMs), exist for most operating systems, including UNIX, the Macintosh OS, and Windows. Bytecode can also be converted directly into machine language instructions by a just-in-time compiler (JIT).

Servlet: An applet that runs on a server. The term usually refers to a Java applet that runs within a Web server environment. This is analogous to a Java applet that runs within a Web browser environment. Java servlets are becoming increasingly popular as an alternative to CGI programs. The biggest difference between the two is that a Java applet is persistent. Once it is started, a servlet stays in memory and can fulfill multiple requests. In contrast, a CGI program disappears once it has fulfilled a request. The persistence of Java applets tends to make them faster.

XML: Short for *Extensible Markup Language*, a specification developed by the W3C. XML is a pared-down version of SGML, designed especially for Web documents. It allows designers to create their own customized tags, enabling the definition, transmission, validation, and interpretation of data between applications and between organizations. For further description of XML, see, e.g., *Extensible Markup Language (XML) 1.0* specification which is available from the World Wide Web Consortium (www.w3.org), the disclosure of which is hereby incorporated by reference. The specification is also currently available on the Internet at *http://www.w3.org/TR/REC-xml.*

SUMMARY OF THE INVENTION

An order scheduling system providing a method for distributing product orders to multiple fulfillers is described. This method solves the common business problem of scheduling order shipments. The method is both optimal and fair (among multiple otherwise-equal fulfillers). It is optimal because it minimizes the number of orders across fulfillers, thus minimizing shipping costs. It is fair because orders are distributed equally across fulfillers if that fulfiller has the product available.

To schedule orders, a data structure is defined whose rows are represented by a hash table of Fulfillers (HF), where each column is a hash table of Products (HP) and where each index of HP is itself a bit vector (VP_i). This gives a three-dimensional data structure. Here, the term "hash" is used to indicate that for a given fulfiller/product pair, the approach may "hash" (i.e., index on) that pair for indexing into a particular cell in the table. This fulfiller/product correspondence may also be represented by a two-dimensional array (e.g., accessible via numeric indexes). Although the information represented in the hash table may be derived from SQL queries submitted to a product/supplier database, it is more efficient to maintain this information in a relatively terse in-memory data structure, as the information will be repeatedly accessed.

In contrast to using linear equations for representing this information, the hash table itself is extensible. If additional fulfillers or suppliers become available, the number of rows in the hash table is simply increased (or simply decreased to represent less). In a corresponding manner, the number of rows in the hash table may be increased or decreased to accommodate changes in the current product offerings. Thus, when changes occur, as they invariably will, the hash table may readily accommodate those changes. There is no requirement that the underlying program code be modified.

Whether a particular fulfiller has a product available does not necessarily depend on that fulfiller's inventory. Certainly, limited inventory poses a problem to product availability by a fulfiller. However, some products have effectively unlimited inventory. For example, the ability of a photofinisher to provide an almost unlimited number of reprints for a customer photograph is one such example. Here, the photofinisher (fulfiller) has effectively unlimited supply of photo-finishing paper available for completing the customer order (of reprinting a photograph). Therefore, often the issue of whether a particular product is available from a given fulfiller depends on whether that fulfiller actually even offers that product to begin with.

An order itself may be viewed as one or more order items (typically, corresponding to a particular product). In certain cases, it may be necessary to split a customer order among the multiple fulfillers, based on order items. For example, an order may be split into two order items, OI_1 and OI_2, in effect, two suborders. The order may be split by having one fulfiller, F_1, fulfill OI_1, while a second fulfiller, F_2, fulfills OI_2. A VP_i vector, which extends into the third dimension (z axis) of the above hash table, is potentially employed in such instances. The VP_i vector allows tracking of what part of an order -- specifically, what order item -- was split and to which fulfiller.

Using the above-described bit vectors, the method may perform bitwise ANDing (&) operations of the bit vectors. This generates an Order bit vector representing the optimized fulfillment (per system configuration/constraints) for a particular received order.

BRIEF DESCRIPTION OF THE DRAWINGS

Fig. 1 is a graph illustrating the simplex method for scheduling order fulfillment.

Fig. 2 is a block diagram of a computer system in which software-implemented processes of the present invention may be embodied.

Fig. 3 is a block diagram of a software system for controlling the operation of the computer system of Fig. 2.

Fig. 4A is a diagram illustrating a hash table data structure employed in the system of the present invention.

Fig. 4B is a diagram illustrating simple order splitting.

Fig. 4C is a diagram illustrating more-complex order splitting.

Fig. 5A is a high-level block diagram illustrating a server-based order scheduling system of the present invention.

Fig. 5B is a block diagram illustrating an order scheduler module of the order scheduling system.

Fig. 5C is a block diagram illustrating a database schema employed by the order scheduling system.

Fig. 6 is a flowchart illustrating overall operation of the order scheduling system of the present invention.

Fig. 7 is a flowchart illustrating a coarse-grained optimization method for fulfilling an order with a relatively low number of fulfillers.

Fig. 8 is a flowchart illustrating a method for minimizing shipping costs by placing order items with fulfillers geographically nearest to the delivery address.

DETAILED DESCRIPTION OF A PREFERRED EMBODIMENT

The following description will focus on the presently-preferred embodiment of the present invention, which is implemented in an Internet-connected server environment running under a server operating system, such as the Microsoft® Windows XP running on an IBM-compatible PC. The present invention, however, is not limited to any particular one application or any particular environment. Instead, those skilled in the art will find that the system and methods of the present invention may be advantageously embodied on a variety of different platforms, including Macintosh, Linux, BeOS, Solaris, UNIX, NextStep, FreeBSD, and the like. Therefore, the description of the exemplary embodiments that follows is for purposes of illustration and not limitation.

I. Computer-based implementation

A. Basic system hardware (e.g., for desktop and server computers)

The present invention may be implemented on a conventional or general-purpose computer system, such as an IBM-compatible personal computer (PC) or server computer. Fig. 2 is a very general block diagram of an IBM-compatible system 200. As shown, system 200 comprises a central processing unit(s) (CPU) or processor (s) 201 coupled to a random-access memory (RAM) 202, a read-only memory (ROM) 203, a keyboard 206, a pointing device 208, a display or video adapter 204 connected to a display device 205, a removable (mass) storage device 215 (e.g., floppy disk, CD-ROM, CD-R, CD-RW, or the like), a fixed (mass) storage device 216 (e.g., hard disk), a communication port(s) or interface(s) 210, a modem 212, and a network interface card (NIC) or controller 211 (e.g., Ethernet). Although not shown separately, a real-time system clock is included with the system 200, in a conventional manner.

CPU 201 comprises a processor of the Intel Pentium® family of microprocessors. However, any other suitable microprocessor or microcomputer may be utilized for implementing the present invention. The CPU 201 communicates with other components of the system via a bi-directional system bus (including any necessary input/output (I/O) controller circuitry and other "glue" logic). The bus, which includes address lines for addressing system memory, provides data transfer between and among the various components. Description of Pentium-class microprocessors and their instruction set, bus architecture, and control lines is available from Intel Corporation of Santa Clara, CA. Random-access memory 202 serves as the working memory for the CPU 201. In a typical configuration, RAM of sixteen megabytes or more is employed. More or less memory may be used without departing from the scope of the present invention. The read-only memory (ROM) 203 contains the basic input output system code (BIOS) -- a set of low-level routines in the ROM that application programs and the operating systems can use to interact with the hardware, including reading characters from the keyboard, outputting characters to printers, and so forth.

Mass storage devices 215, 216 provide persistent storage on fixed and removable media, such as magnetic, optical or magnetic-optical storage systems, flash memory, or any other available mass storage technology. The mass storage may be shared on a network, or it may be a dedicated mass storage. As shown in Fig. 2, fixed storage 216 stores a body of program and data for directing operation of the computer system, including an operating system, user application programs, driver and other support files, as well as other data files of all sorts. Typically, the fixed storage 216 serves as the main hard disk for the system.

In basic operation, program logic (including that which implements methodology of the present invention described below) is loaded from the storage device or mass storage 216 into the main (RAM) memory 202, for execution by the CPU 201. During operation of the program logic, the system 200 accepts user input from a keyboard 206 and pointing device 208, as well as speech-based input from a voice recognition system (not shown). The keyboard 206 permits selection of application programs, entry of keyboard-based input or data, and selection and manipulation of individual data objects displayed on the display screen 205. Likewise, the pointing device 208, such as a mouse, track ball, pen device, or the like, permits selection and manipulation of objects on the display screen. In this manner, these input devices support manual user input for any process running on the system.

The computer system 200 displays text and/or graphic images and other data on the display device 205. The video adapter 204, which is interposed between the display 205 and the system, drives the display device 205. The video adapter 204, which includes video memory accessible to the CPU 201, provides circuitry that converts pixel data stored in the video memory to a raster signal suitable for use by a cathode ray tube (CRT) raster or liquid crystal display (LCD) monitor. A hard copy of the displayed information, or other information within the system 200, may be obtained from the printer 207, or other output device. Printer 207 may include, for instance, an HP LaserJet® printer (available from Hewlett-Packard of Palo Alto, CA), for creating hard copy images of output of the system.

The system itself communicates with other devices (e.g., other computers) via the network interface card (NIC) 211 connected to a network (e.g., Ethernet network), and/or modem 212 (e.g., 56K baud, ISDN, DSL, or cable modem), examples of which are available from 3Com of Santa Clara, CA. The system 200 may also communicate with local occasionally-connected devices (e.g., serial cable-linked devices) via the communication ("comm") interface 210, which may include a RS-232 serial port, a Universal Serial Bus (USB) interface, or the like. Devices that will be commonly connected locally to the interface 210 include laptop computers, handheld organizers, digital cameras, and the like.

IBM-compatible personal computers and server computers are available from a variety of vendors. Representative vendors include Dell Computers of Round Rock, TX, Compaq Computers of Houston, TX, and IBM of Armonk, NY. Other suitable computers include Apple-compatible computers (e.g., Macintosh), which are available from Apple Computer of Cupertino, CA, and Sun Solaris workstations, which are available from Sun Microsystems of Mountain View, CA.

B. Basic system software

Illustrated in Fig. 3, a computer software system 300 is provided for directing the operation of the computer system 200. Software system 300, which is stored in system memory (RAM) 202 and on fixed storage (e.g., hard disk) 216, includes a kernel or operating system (OS) 310. The OS 310 manages low-level aspects of computer operation, including managing execution of processes, memory allocation, file input and output (I/O), and device I/O. One or more application programs, such as client application software or "programs" 301 (e.g., 301a, 301b, 301c, 301d) may be "loaded" (i.e., transferred from fixed storage 116 into memory 102) for execution by the system 100.

System 300 includes a graphical user interface (GUI) 315, for receiving user commands and data in a graphical (e.g., "point-and-click") fashion. These inputs, in turn, may be acted upon by the system 100 in accordance with instructions from operating system 310, and/or client application module(s) 301. The GUI 315 also serves to display the results of operation from the OS 310 and application(s) 301, whereupon the user may supply additional inputs or terminate the session. Typically, the OS 310 operates in conjunction with device drivers 320 (e.g., "Winsock" driver -- Windows' implementation of a TCP/IP stack) and the system BIOS microcode 330 (i.e., ROM-based microcode), particularly when interfacing with peripheral devices. OS 310 can be provided by a conventional operating system, such as Microsoft® Windows 9x, Microsoft® Windows NT, Microsoft® Windows 3000, or Microsoft® Windows XP, all available from Microsoft Corporation of Redmond, WA. Alternatively, OS 310 can also be an alternative operating system, such as the previously- mentioned operating systems.

The above-described computer hardware and software are presented for purposes of illustrating the basic underlying desktop and server computer components that may be employed for implementing the present invention. For purposes of discussion, the following description will present examples in which it will be assumed that there exists at least one host computer (e.g., "server") that may communicate with one or more other computers (e.g., "clients"). The present invention, however, is not limited to any particular environment or device configuration. Instead, the present invention may be implemented in any type of system architecture or processing environment capable of supporting the methodologies of the present invention presented in detail below.

II. Order scheduler

A. Introduction

Existing order fulfillment systems typically include a database of products (that are being sold to customers), with the database tracking which product can be fulfilled or supplied by which fulfiller(s). Thus, an existing one-to-many relationship of product to fulfiller/supplier is often already tracked in conventional fulfillment systems. For a given row (record) of product represented in the database, a given supplier either has or does not have that product; this may be represented as a simple Boolean (true/false) value. In the aggregate (i.e., for multiple fulfillers), availability by fulfiller may be represented by a bit vector (i.e., a one-dimensional array of bit values), where a bit value of 1 represents true (i.e., the fulfiller has the product) and a bit value of 0 represents false (i.e., the fulfiller does not have the product). By using bit vectors to represent this information (which itself may be extracted from a product/fulfiller database), the task of programming an appropriate solution is simplified.

B. Overview

The following description presents a method for distributing product orders to multiple fulfillers. This method solves the common business problem of scheduling order shipments. The method is both optimal and fair (among multiple otherwise-equal fulfillers). It is optimal because it minimizes the number of orders across fulfillers, thus minimizing shipping costs. It is fair because orders are distributed equally across fulfillers if that fulfiller has the product available.

To schedule orders, a data structure is defined whose rows are represented by a hash table of Fulfillers (HF), where each column is a hash table of Products (HP) and where each index of HP is itself a vector (VP_i). This gives the three-dimensional data structure shown in Fig. 4A. Here, the term "hash" is used to indicate that for a given fulfiller/product pair, the approach may "hash" (i.e., index on) that pair for indexing into a particular cell in the table. This fulfiller/product correspondence may also be represented by a two-dimensional array (e.g., accessible via numeric indexes). Although the information represented in the hash table may be derived from SQL queries submitted to a product/supplier database, it is more efficient to maintain this information in a relatively terse in-memory data structure, as the information will be repeatedly accessed.

In contrast to using linear equations for representing this information, the hash table itself is extensible. If additional fulfillers or suppliers become available, the number of rows in the hash table is simply increased (or simply decreased to represent less). In a corresponding manner, the number of rows in the hash table may be increased or decreased to accommodate changes in the current product offerings. Thus, when changes occur, as they invariably will, the hash table may readily accommodate those changes. There is no requirement that the underlying program code be modified.

Whether a particular fulfiller has a product available does not necessarily depend on that fulfiller's inventory. Certainly, limited inventory poses a problem to product availability by a fulfiller. However, some products have effectively unlimited inventory. For example, the ability of a photofinisher to provide an almost unlimited number of reprints for a customer photograph is one such example. Here, the photofinisher (fulfiller) has effectively unlimited supply of photo-finishing paper available for completing the customer order (of reprinting a photograph). Therefore, often the issue of whether a particular product is available from a given fulfiller depends on whether that fulfiller actually even offers that product to begin with.

An order itself may be viewed as one or more order items (typically, corresponding to a particular product). In certain cases, it may be necessary to split a customer order among the multiple fulfillers, based on order items. Fig. 4B illustrates this concept. There, the order has been split into two order items, OI_1 and OI_2, in effect, two suborders. As shown, the order has been split by having one fulfiller, F_1, fulfill OI_1, while a second fulfiller, F_2, fulfills OI_2. The VP_i vector, which extends into the third dimension (z axis), is potentially employed in such instances. The VP_i vector allows tracking of what part of an order -- specifically, what order item -- was split and to which fulfiller it was split to. Fig. 4C illustrates this concept. Suppose an order consists of three order items: a 5x7 reprint of a dog photograph (product type of P_1), a 5x7 reprint of a cat photograph (also product type of P_1), and a coffee mug embossed with a bird photograph (product type of P_2). To track that fulfiller F_1 is to fulfill the order items of type P_1, OI_1 and OI_2 are entered in the VP_i vector that is indexed by F_1 and P_1.

C. Example: Setting up bit vectors

To understand how the order scheduler methodology works, consider the following example, which illustrates use of bit vectors. An order may contain any or all of the following products.

P_1 4x6 photographic print

P_2 8x10 photographic print

P_3 coffee mug embossed with photo

The fulfillment system will have one or more fulfillers that can be chosen to fill the order. However, not every fulfiller may have all products.

F_1 4X6 print, coffee mug (2 products)

F_2 coffee mug (1 product)

F_3 4x6 print, 8x10 print, coffee mug (3 products)

The function `F1.count()` returns 2.

In order to represent the products a fulfiller can supply, a bit vector is used, as follows:

F_1.bv[101] (which is P1 and P3)

F_2.bv[001] (which is P3)

F_3.bv[111] (which is P1, P2 and P3)

An Order is a sequence of OrderItems where each OrderItem is one product. An Order could appear as:

Order

OrderItem 4x6 print

OrderItem coffee mug

The above order would have the following bit vector `Order.bv[101]`. Initially, the method may save the enumeration of the fulfiller hash table in another vector called `AllF [`F_1 F_2 F_3`]`. The sequence of the fulfillers in this vector is subject to change. All of the fulfillers may be saved in another vector and ordered by the number of products they fulfill; this is called vector `AllFByCount [`F_3 F_1 F_2`]`. F_3 has the most products so it appears first in the vector. The sequence of this vector is fixed and will not change. As described in further detail below, the basic approach adopted includes performing bitwise ANDing operations of the bit vectors, to generate an Order bit vector representing the optimized fulfillment (per system configuration/constraints) for a particular received order.

D. Order Scheduler components

As shown in Fig. 5A, a server-based order scheduler system 500 constructed in accordance with the present invention includes an order parser 510, an order engine 520, an order scheduler 530, an order processing module 540, and a back-end (SQL) database 550. The order parser 510 receives XML requests (e.g., via HTTP, FTP, or the like) that comprise order information for orders that are to be fulfilled. The parsed information, in turn, is stored in the database 550 as an order (comprising one or more order items) in `ORDER_T` and `ORDER_ITEM_T` tables. The order engine 520 may now invoke various components for handling the database-logged order. In particular, the engine 520 invokes the order scheduler 530, described in further detail below, for optimizing fulfillment of each received order. Once the desired optimized fulfillment of a particular order has been determined, the order processing module 540 is invoked for actual processing of the order, such as submitting online fulfillment requests (i.e., order to particular fulfillers) and/or generating hardcopy documents, as desired.

Of particular interest is the order scheduler 530, which is illustrated in greater detail in Fig. 5B. As shown, the order scheduler 530 includes a scheduler core or engine 531, a database interface 533, and a configuration/constraints module 535. These modules are implemented as server-side Java components (servlets). The scheduler core 531 implements the core logic for optimizing fulfillment of orders; its detailed operation is discussed below. The database interface 533 provides databases connectivity (including SQL query ability) to the back-end database. The configuration/constraints module 535 allows specification of various policies, including configuration settings and fulfillment constraints. For example, the module 535 may be employed to specify a fulfillment constraint that a particular fulfiller/supplier is most favored. Additionally, the module 535 allows specification that orders are to be fulfilled by proximity (of fulfiller to customer), by count (of order items per fulfiller), or the like.

Fig. 5C illustrates construction of the database 550 in further detail. The database itself may be implemented as an SQL-based relational database, such as an Oracle database (e.g., in Oracle 8i, available from Oracle Corporation of Redwood Shores, CA). In particular, the figure demonstrates a database schema employed for the database 550 in the currently-preferred embodiment. The following database tables are of particular interest in the database schema:

ORDER_T -- describes an order, has a set of order items;

ORDER_ITEM_T -- describes what the product and order item specialty is that a user ordered;

FULFILLER_ORDER_T -- allows splitting an order into multiple fulfiller orders (sub-orders);

FULFILLER_ORDER_ITEM_T -- similar to ORDER_ITEM_T but represents the order items in a sub-order;

PRODUCT_T -- describes a product, e.g., a 4x6 photo print, or the like; and

PRODUCT_FULFILLER_T -- associates which products are available for which fulfillers.

E. Detailed Operations

1. High-level method of operation

Fig. 6 is a flowchart summarizing overall operation of the system. As shown at step 600, the bit vectors mapping product availability for each fulfiller are routinely updated from the database. This updating includes adding/deleting fulfillers to a vector of fulfillers. The updating may include refreshing each bit according to periodic updates in inventory tables, in cases of extended implementations of the preferred embodiment that incorporate inventory data. During the update, each fulfiller is allotted a bit vector with a matching number of elements as the product hash table has cells corresponding to all the products (P_1, P_2, ... P_N) provided by the middleman. A new vector is built for each fulfiller, wherein each element, or bit, in the vector (which corresponds to each product type of the middleman, in the product hash table) is set to 1, for true, if the fulfiller provides that product. Otherwise, that bit is set to 0, for false if the fulfiller does not provide this type of product. Therefore, if the middleman offers three types of products, and a particular fulfiller does provides the second and third products (as represented by the product hash table) the fulfiller's bit vector would be set to "011".

At step 610, the middleman sets the scheduling policy for determining the selection criteria for the fulfiller precedence, or sort-ordering in a vector of all the fullfillers' bit vectors. In the most general case, the selection criteria, or constraint, is to minimize the number of fulfillers splitting the fulfillment of each individual middleman order. The sort-ordering of a vector of all fulfillers reflects the selection criteria by advancing the fulfiller whose most satisfies that criteria to the front of a runtime vector of fulfillers' bit vectors, `AllF[].bv`. When a fulfillment order is scheduled, the fulfiller vector is processed sequentially from element 0 to element N-1. Fulfillers at the front of the fulfiller vector have an earlier opportunity to fulfill each order item in a fulfillment order.

At step 620, the order scheduler engine (described above) fetches a fulfillment order request. The order parser had already populated the database with the order data when the XML-encoded order request was received by either an HTTP or an FTP communication with the order-generating client (e.g., order entry at a PC). At step 630, the order items requested in the fulfillment order are mapped to the product items in the middleman's product line, the product hash table. Using the same bit vector mapping for the order's bit vector as was used at step 610 (in creating each fulfiller's bit vector: order items whose product type matched one of the middleman's product types), at step 630, an order bit vector is populated for the current order. Using the same logic as was used at step 610, at step 630, a new order vector is built for the current order, wherein each element, or bit, in the vector (which corresponds to each product type of the middleman, in the product hash table) is set to 1, for true, if the order requests that product. Otherwise, that bit is set to 0, for false if the order does not request this type of product. Therefore, if the middleman offers three types of products, and a particular order requests the first and second products (as sequenced in the product hash table) the fulfiller's bit vector would be set to "110".

At step 640, the order scheduler 430 walks through the fulfiller vector, executing a bitwise & ("AND") operation on the current fulfiller bit vector and the order bit vector. If the bitwise & operation on these two bit vectors results in a bit vector with the same bit sequence as the order bit vector, then step 640 successfully fulfills the entire order with a single fulfiller (the most optimal condition), exits, and jumps ahead to step 670. If a single fuller cannot fulfill an entire order, then the order fulfillment must be split across multiple fulfillers; the method proceeds to the next step. At step 650, the order scheduler 430 walks through the sort-ordered fulfiller vector, traverses each fulfiller's product vector, and executes a bitwise & ("AND") operation on the current fulfiller bit vector and an order item bit vector for each order item in the order bit vector. While the order bit vector is a "bitwise OR" of all the order items, the order item bit vector has only 1 bit set. For example, if the order bit vector is represented by "101", the order item bit vector for the third product would be "001". The order item bit vector is a temporary structure used to determine where the order items should be placed in the three-dimensional data structure shown in Fig. 4A. In this step the order scheduler 430 places order items with multiple fulfillers, rather than placing the whole order with a single fulfiller. The following Java snippet tests an order item with the types of products offered by the earliest selected fulfiller:

```
(AllFSortedForAscendingProver[0].bv & OrderItem.bv) == OrderItem.bv
```

If the OrderItem cannot be placed in `AllFSortedForAscendingProver [0]` then the order scheduler 430 tries the next index, `AllFSortedForAscendingProver [1]` and so on until this order item is placed with a fulfiller. This iteration is run against every order item sequentially in a single pass through a sort-ordered list of fulfiller.

The preferred embodiment depends upon a statistically likely optimization of a minimum set of fulfillers for an order; therefore, computation time is minimal. The base program module is implemented for the present invention, thereby freeing programmatic resources for the middleman, and it need not be modified. However, it does serve as an implementation design that application developers can use to further refine their scheduling algorithms. At step 650, ancillary constraints may be accounted for by re-sort-ordering the fulfillers' vector (i.e., a fulfiller preference vector). For example, to implement another selection criteria, such as "favored fulfiller", the index position(s) for the favored fulfiller(s) can be re-indexed to the beginning of the fulfillers' vector.

At step 660, as the order items among the split order are being placed with providing fulfillers, the order scheduler 430 populates a subsequent dimension (VP_i, described above) for the two-dimensional matrix mapping fulfiller products with the middleman's product line. This third, or higher, dimension represents collections (vectors) of order requests for the same type of product offering, but with subtler typing information in the description of an order item. For example, using an e-commercial photographic printing service as a fictitious fulfiller or middleman, three types of products are offered: 4x6 prints, 8x10 prints, and 11x14 prints. The product type, 4x6 prints, is represented by a cell in the 2-dimensional hash tables matrix (as previously described). However, an order may list a 4x6 print as two separate order items: one 4x6 print might be of the owner's dog, whereas the subsequent 4x6 print might be of the owner's cat. In this example scenario, both 4x6 order items would be placed with the same fulfiller, who provides 4x6 prints, and each placement would be recorded in a separate element along the VP_i vector indexed at the 4x6 product type for the current fulfiller.

When scheduling a split order, the creation of a new set of fulfillment orders (to be placed with multiple fulfillers) that contains only a subset of the order items in the original customer order, is termed a sub-order. At step 660 the fulfillers vector is walked, nesting another walk along the fulfiller's product bit vector, and at each index in the 2-dimensional matrix traverse along the corresponding VP_i vector, gathering order items placed along the VP_i vector to be placed in each provider's sub-order.

Step 660 is the final step necessary to schedule an order fulfillment. However, before re-cycling through the sequence envisioned in Fig. 6 for fulfilling another order, the middleman may desire to re-order the sort-ordered fulfiller vector. At step 670, ancillary constraints may be accounted for by modifying the sort-ordered fulfiller vector, or fulfiller preference vector, outside of the execution loop that processes the scheduling of the fulfillment of each order. For example, if "fulfiller placement fairness" were an additional selection criteria (including the minimum number of fulfillers criteria), then subsequent to fulfilling an order, the elements in the sort-ordered fulfiller vector move one index closer to the front of the vector (e.g., `AllF[0]`) to enable round-robin fulfiller preference.

2. Probable Fulfillers

Fig. 7 is a flowchart illustrating a coarse-grained optimization method for fulfilling an order with a relatively low number of fulfillers. The underlying model for this single-pass method is that sort-ordering ranking fulfillers according to the number of order items in the current order each fulfiller can place or "fulfill," results in a statistically improved likelihood of scheduling this order with a lower-than-average number of fulfillers. At step 700, each fulfiller from a fulfiller vector gets the number of order items it can place, with a method, `AllF[].count()`. At step 720, the vector of fulfillers, `AllF[]`, is sorted, in descending order, according to the number of order items that fulfiller can place. At steps 640 and 650, these steps proceed as previously described for steps 640 and 650 in Fig. 6. The method, described in Fig. 7, minimizes shipping costs by minimizing the number of disparate fulfillers, or sub-orders, each of which would likely attach a levy for basic services rendered.

3. Minimal shipping distance

Figs. 8 is a flowchart illustrating the methodology of the preferred embodiment processing arbitrary logic to arrive at an optimized fulfillment schedule for an order. Fig. 8 describes the middleman method for minimizing shipping costs by placing order items with fulfillers geographically nearest to the delivery address. Minimizing shipping distance lowers shipping costs. At step 800, the geographical proximity of each fulfiller to the delivery address is calculated. This method utilizes the zip code prefixes and corresponding postal zones as devised by the United States Postal Service. These prefixes typically comprise the three most significant digits of a zip code and define rather distinct geographical locations. Since localities in the United States are parceled into postal zones designated by zip codes, utilization of zip code prefixes readily provides a means for identifying any geographical location in the country.

At step 820, a method ranks fulfillers according to their proximity to the delivery address of a sub-order shipment. The fulfiller vector, `AllFByProximity[]`, is sort-ordered from the least distant shipping route (to the delivery address) to the most distant. The proximity values are calculated as follows. First (for the continental United States), the numeric shipping zone (i.e., the above-mentioned United States Postal Service-designated shipping zones) of the delivery address is calculated using that address's zip code. The numeric shipping zone of the each fulfiller was already fetched from the database. Now the two endpoints of a shipping route, both the fulfiller and the delivery address, are represented by an integer in the set of shipping zones, 0-9. Adjacent zones have successive zone values. Second, the numeric difference between the value of the delivery address shipping zone and the value of a fulfiller's shipping zone is the fulfiller's proximity value. The lower that numeric difference, the greater the fulfiller's proximity. For example, addresses in zone 2 have the highest proximity to fulfillers also in zone 2. Their proximity values would be 0. The next highest proximity for addresses in zone 2 would be both zone 1 and zone 3. Third, in determining the proximity value for a fulfiller in another zone, the value of the shipping zone of the delivery address is both incremented/decremented to test for "next best proximity." In this example, the fulfiller's shipping zone value (2) would be compared with zone 1 and then with zone 3. Fourth would be an iteration of the previous step, but the already incremented/decremented delivery address zones are yet again incremented/decremented. In this example, the fulfiller's shipping zone value (2) would be compared with zone 0 and then with zone 4.

At step 640, the sort-ordered vector of fulfillers, `AllFByProximity[]`, is tested to determine if the order can be fulfilled by a single fulfiller: Each fulfiller bit vector is processed with the order bit vector with a "bitwise &" operation to place order items with fulfillers preferred by their proximity. At step 650, the sort-ordered vector of fulfillers' bit vectors, `AllFByProximity[].bv`, is processed as previously described at step 650 in Fig. 6. The fulfillers' vector is created by a `byProximity()` method.

The `byProximity()` method itself may be implemented as follows (shown in Java code).

```
// Return a list of fulfillers ordered by those closest to this
zipCode
  public Vector byProximity (String zipCode) {
    int zoneOfZipCode;
    Vector vectorOfFulfillers = new Vector();
    Vector fulfillersTmp;
    int step = 1;
    boolean keepGoing;
    int i;

    // The first digit of a zip code is the "national area" of the
country.
    // The areas are:
    //   0 Northeast
    //   1 NewYork
    //   2 MidAtlantic
    //   3 Southeast
    //   4 GreatLakes
    //   5 Midwest
    //   6 Plains
    //   7 Southwest
    //   8 Western
    //   9 Pacific
    // This information is not online and I derived it by looking at
post office
    // maps. So the names may not be correct but it is close enough
for postal work.

    try {
        zoneOfZipCode = Integer.parseInt(zipCode.substring(0, 1));
    } catch(Exception e) {
        return vectorOfFulfillers; // passed in a malformed zip code
    }
    fulfillersTmp = Fulfiller.getByZone(zoneOfZipCode);
    for (i = 0; i < fulfillersTmp.size(); i++)
32:        vectorOfFulfillers.addElement(fulfillersTmp.elementAt(i));
33:
34:     while (true) {
35:       keepGoing = false;
36:
37:       // we may get a zone in the middle of the country so we need to
step
38:       // away 1 zone at a time to make sure that we get the fulfillers
closest
39:       // to this zone
40:
41:       if (zoneOfZipCode + step <= Fulfiller.LAST_ZIP_ZONE) {
42:         fulfillersTmp = Fulfiller.getByZone(zoneOfZipCode + step);
43:         for (i = 0; i < fulfillersTmp.size(); i++)
44:            vectorOfFulfillers.addElement(fulfillersTmp.elementAt(i));
45:         keepGoing = true;
46:       }
47:
48:       if (zoneOfZipCode - step >= Fulfiller.FIRST_ZIP_ZONE) {
49:         fulfillersTmp = Fulfiller.getByZone(zoneOfZipCode - step);
50:         for (i = 0; i < fulfillersTmp.size(); i++)
51:            vectorOfFulfillers.addElement(fulfillersTmp.elementAt(i));
52:         keepGoing = true;
53:       }
54:
55:       if (keepGoing == true)
56:         step++;
57:       else
58:         break;
59:
60:     } // end while true
61:
62:     return vectorOfFulfillers;
63:   }
64:
```

As shown by lines 41 and 48, the method builds a list of who are the fulfiller(s) in the immediate and then neighboring ZIP code regions. The method proceeds to add fulfillers from other ZIP code regions, proceeding from nearest to farthest (from the customer), until all fulfillers have been ranked by proximity.

Appended herewith as Appendix A are source code listings in the Java programming language, providing further description of the present invention. A suitable development environment for compiling Java code is available from a variety of vendors, including Borland Software Corporation (formerly, Inprise Corporation) of Scotts Valley, CA and Sun Microsystems of Mountain View, CA. Appendix A is hereby incorporated by reference.

While the invention is described in some detail with specific reference to a single-preferred embodiment and certain alternatives, there is no intent to limit the invention to that particular embodiment or those specific alternatives. For instance, those skilled in the art will appreciate that modifications may be made to the preferred embodiment without departing from the teachings of the present invention.

APPENDIX A: SOURCE CODE LISTINGS

```
/**************************************************************************/
/*  ProjectName :   ePhoto.com
    */
/*  ModuleName  :   ec.fulfillment                                         */
/*  File        :   DistributeOrders.java                                  */
/*                                                                         */
/*  Copyright 1998-2000 by LightSurf Technologies.,                        */
/*  110, Cooper Street, Santa Cruz, CA-95060, U.S.A.                       */
/*  All rights reserved.                                                   */
/*                                                                         */
/*  This software is the confidential and proprietary information          */
/*  of LightSurf Technologies. ("Confidential Information").               */
/*  You shall not disclose such Confidential Information and shall use     */
/*  it only in accordance with the terms of the license agreement          */
/*  you entered into with LightSurf Technologies.                          */
/**************************************************************************/
package com.lightsurf.ec.fulfillment;

import java.util.*;
import java.sql.*;
//import com.lightsurf.API.*;
import com.lightsurf.util.Config;
import com.lightsurf.ec.API.Table;
import com.lightsurf.ec.API.DBContext;
import com.lightsurf.ec.API.Persistance;
import com.lightsurf.services.ecomm.Order;
import com.lightsurf.services.ecomm.OrderItem;
//import com.lightsurf.ec.API.OrderTable;
import com.lightsurf.services.ecomm.Address;
import com.lightsurf.services.ecomm.Product;
import com.lightsurf.services.ecomm.EcommService;
import com.lightsurf.ec.fulfillment.FulfillerOrder;
import javax.servlet.ServletContext;
import com.lightsurf.services.ecomm.EcommService;

/**
 * This class looks through the order table for new orders and rewrites
them as
 * FulfillerOrders. The FulfillerOrder maintains similar links to Order
but is
 * much smaller. An Order can have several different shipping addresses.
The
 * FulfillerOrder only has 1 shipping address. This class splits 1 order
into
 * 1 or more FulfillerOrder objects by unique address.
 *
 * @author John Rodriguez
 * @version 0.1
 */

public class DistributeOrders extends Persistance
{
  static final boolean debug = false;

  // Currently the bit vector size is 64 (long). This limits the scheduler
to 64 products
  // and 64 fulfillers. This assumption has sped up both the
implementation of the
  // scheduler as well as it's performance. Actually, the bit vector could
be any
```

```
  // arbitrary length. If 64 is too restrictive, I will write a general
bit vector
  // package that handles arbitrary number of products or fulfillers.

  // There could be more scheduling modes than the following 2 modes. For
now, these
  // 2 should show what needs to be done for 2 different scheduling modes.
  final static public int SCHEDULE_BY_COUNT = 0;
  final static public int SCHEDULE_BY_PROXIMITY = 1;
  final static public int SCHEDULE_MODES_MAX = SCHEDULE_BY_PROXIMITY;

  // This object is only used for the monitor of the wait/notify
  Integer syncObj = new Integer(1);

  // If we were started by a servlet, remember the context for logging.
  ServletContext context = null;

  // Save all the fulfillers.
  private Vector fulfillers = null; // for HF
  // Save all the products
  private Product[] products = null;   // for HP

  // The schedulerTable is a data structure whose rows are represented by
a
  // HashTable of fulfillers (HF), where each column is a hashtable of
products (HP)
  // and where each indice of HP is itself a vector (VPi). For more detail
on
  // how these data structures are used, see the document "Order
Scheduler" by
  // John Rodriguez
  Hashtable HF;
  Hashtable HP;
  Vector VPi;

  // This vector of all the fulfillers changes as we distribute orders
  Vector AllF = new Vector();
  Vector AllFByCount = new Vector();
  Vector AllFByProximity = new Vector();

  EcommService service = EcommService.SYSTEM_ECOMM_SERVICE;

  public DistributeOrders(ServletContext context) {
    int i, j;
    Fulfiller f, fTmp;
    int productCount;
    Vector productFulfillers;
    ProductFulfiller productFulfiller;
    Product prod;
    int numberOfProducts;

    this.context = context;

    try {
        products = service.selectProducts();
        numberOfProducts = products.length;

        ProductFulfiller.init();
        productFulfillers = ProductFulfiller.getAllProducts();
        Fulfiller.init();
        fulfillers = Fulfiller.getAllFulfillers();

        HF = new Hashtable(fulfillers.size());
        // Build the main data structure for scheduling
        for (i = 0; i < fulfillers.size(); i++) {
        HP = new Hashtable(numberOfProducts);

        f = (Fulfiller)fulfillers.elementAt(i);
        AllF.addElement(f);
        // Build the AllFByCount vector in descending product count order
        if (AllFByCount.size() == 0) {
            AllFByCount.addElement(f);
        }
        else {
            AllFByCount.insertElementAt(f, 0);
            // Build the AllFByCount in place rather than sorting later
            for (j = 1; j < AllFByCount.size(); j++) {
            fTmp = (Fulfiller)fulfillers.elementAt(j);
            if (f.getProductCount() < fTmp.getProductCount()) {
                // Switch places
                AllFByCount.insertElementAt(f, j);
                AllFByCount.insertElementAt(fTmp, j-1);
            }
            else {
                break;
            }
            } // end for
        }

        for (j = 0; j < products.length; j++) {
            prod = products[j];
            if (debug)
            System.out.println("productFulfiller product_id " +
prod.getProductID());
            if (HP.containsKey(new Long(prod.getProductID())) == false) {
            VPi = new Vector();
            HP.put(new Long(prod.getProductID()), VPi);
            }
        }
        HF.put(new Integer(f.hashCode()) , HP);
        }

        // Build the bitPattern manipulation data structure. I have chosen
a long
        // for efficiency. This means that the number of products must be
less than 64
        // and the number of fulfillers must be less than 64. I could
write a general
        // bit vector package that would allow for any number of products
or fulfillers
        // but it would be slower. I think in the general case that 64
should be enough.
        // If this proves not to be the case, then I will write the
general bit vector
        // package later.
        // Notice that the bit vectors are indexed according to the
ProductFulfiller table
        // and not the Product table.
        ProductFulfiller product;
        long bv;
        for (i = 0; i < products.length; i++) {
        prod = products[i];
        bv = 1 << i;
        if (debug)
            System.out.println("productFulfillers size " +
productFulfillers.size());
        for (j = 0; j < productFulfillers.size(); j++) {
                product = (ProductFulfiller)productFulfillers.elementAt(j);
                if (debug)
                    System.out.println("product pfulfiller id " +
prod.getProductID() + " " + product.getProductID());
                if (prod.getProductID() == product.getProductID()) {
                product.orBV(bv);
                }
            }
            }

            if (debug) {
            for (j = 0; j < productFulfillers.size(); j++) {
                product = (ProductFulfiller)productFulfillers.elementAt(j);
                    System.out.println("pfulfiller " +
product.getFulfillerID() + " " + product.getBV());
            }
            }

            // Copy the bit pattern just set above to the fulfiller fulfiller.
            // A fulfiller has a row for each product they carry in the
ProductFulfiller table.
              for(i = 0; i<fulfillers.size(); i++)
              {
                    Fulfiller fulfiller = (Fulfiller)fulfillers.elementAt(i);
                    for (j = 0; j < productFulfillers.size(); j++ ) {
                    product =
(ProductFulfiller)productFulfillers.elementAt(j);
                    if (debug)
                        System.out.println("product fulfiller id " +
product.getFulfillerID() + " " + fulfiller.getFulfillerID());
                    if (product.getFulfillerID() ==
fulfiller.getFulfillerID()) {
                      fulfiller.orBV(product.getBV()); // this represents the
list of products
                  }
                    } // end for j
              }

            if (debug) {
            for (j = 0; j < fulfillers.size(); j++) {
                Fulfiller fulfiller = (Fulfiller)fulfillers.elementAt(j);
                    System.out.println("fulfiller bv " + fulfiller.getBV());
            }
            }
        } catch (Exception e) {
           if((Config.ERROR_LEVEL & Config.DEBUG_FULFILLMENT) > 0 )
            System.err.println("[error_fulfillment] DistributeOrders:
exception during initialization");
           e.printStackTrace();
        }
    }

    // Find all new orders and add rewrite the Order as a FulfillerOrder
    public void distributeNewOrders(Vector fulfillerOrders, int
schedulingMode) {

        if (debug)
          System.out.println("Start distributeNewOrders");

        if (schedulingMode < DistributeOrders.SCHEDULE_BY_COUNT ||
            schedulingMode > DistributeOrders.SCHEDULE_MODES_MAX) {
            System.err.println("distributeOrders: bad scheduling mode " +
schedulingMode);
            return;
    }

    Vector fulfillerOrderItems;
    Order order;
    OrderItem orderItem;
    OrderItem orderItem1 = null;
    FulfillerOrder fOrder = null;
    FulfillerOrderItem fOrderItem = null;
    int i, j, k;
    Address shippingAddress;
    Long longVal, bitVal;
    Fulfiller fulfiller;
    ProductFulfiller productFulfiller;
    boolean orderDone = true;
    Vector saveNewOrders = new Vector();

    if (debug)
      System.out.println("distributeNewOrders start");

    if (fulfillerOrders != null) {

         for (i = 0; i < fulfillerOrders.size(); i++) {

           fOrder = (FulfillerOrder)fulfillerOrders.elementAt(i);

           fulfillerOrderItems = fOrder.getItems();

           // This initializes the Order and OrderItem for later
comparisons.
           for (j = 0; j < fulfillerOrderItems.size(); j ++) {
              // Need to compare against product id
              fOrderItem =
(FulfillerOrderItem)fulfillerOrderItems.elementAt(j);
              order = fOrder.getParentOrder();
              orderItem = order.getOrderItem(fOrderItem.getOrderItemID());
              Vector pFs =
ProductFulfiller.getProduct(fOrderItem.getFulfillerID());
              for (k = 0; k < pFs.size();k++) {
                  productFulfiller = (ProductFulfiller)pFs.elementAt(k);
                  if (orderItem.getProductID() ==
productFulfiller.getProductID()) {
                    // An Order will have all the products or'ed in.
                    fOrder.orBV(productFulfiller.getBV());
                    // An OrderItem will have just the bit for this
product or'ed in.
                    fOrderItem.orBV(productFulfiller.getBV());
                  }
              }
           }

           // Get the first fulfiller
           fulfiller = (Fulfiller)AllF.elementAt(0);

           orderDone = false;
           if (schedulingMode == DistributeOrders.SCHEDULE_BY_COUNT &&
             (fulfiller.getBV() & fOrder.getBV()) == fOrder.getBV()) {
             long fid = fulfiller.getFulfillerID();
             fOrder.setFulfillerID(fid);
             for (j = 0; j < fulfillerOrderItems.size(); j ++) {
                fOrderItem =
(FulfillerOrderItem)fulfillerOrderItems.elementAt(j);
                fOrderItem.setFulfillerID(fid);
             }
             // we are done with this order
```

```
                fOrder.setStatus(FulfillerOrder.STATUS_PENDING);
                orderDone = true;
              }

              if (orderDone == false) {

                 Vector newFulfillerOrders;

                 // This will return a Vector of 2 or more FulfillerOrders.
These were
                 // split from the original FulfillerOrder.
                 newFulfillerOrders = ScheduleOrders(fOrder, schedulingMode);

                 if (newFulfillerOrders != null) {
                     for (k = 0; k < newFulfillerOrders.size(); k++)

saveNewOrders.addElement(newFulfillerOrders.elementAt(k));
                     newFulfillerOrders.removeAllElements();
                     // Get rid of this order because it has been split
                     fulfillerOrders.removeElementAt(i); i--;
                 }

              }

              if (schedulingMode == DistributeOrders.SCHEDULE_BY_COUNT) {
                // change the sequence of fulfillers in AllF
                Object obj = AllF.elementAt(0);
                AllF.removeElementAt(0);
                AllF.addElement(obj);
              }

            }  // end for i

         if (debug) {
           for (i = 0; i < saveNewOrders.size(); i++)
             System.out.println(i + "fid=" +
((FulfillerOrder)saveNewOrders.elementAt(i)).getFulfillerID());
         }

           for (i = 0; i < saveNewOrders.size(); i++)
             fulfillerOrders.addElement(saveNewOrders.elementAt(i));

         if (debug) {
           for (i = 0; i < fulfillerOrders.size(); i++)
             System.out.println(i + "fid=" +
((FulfillerOrder)fulfillerOrders.elementAt(i)).getFulfillerID());
         }

           saveNewOrders.removeAllElements();

      } // end fulfillerOrders != null
    } // end processNewOrders

    // Split is given a FulfillerOrder that could not be scheduled with the
first
    // fulfiller on the list. It will have to be broken up across multiple
fulfillers.
    // Pass back a vector of newly split FulfillerOrders. We use the HF, HP
and VPi
    // to see if we can place the FulfillerOrderItems in a slot.
    // If the fOrder was not able to be split, this is an error. Log it and
pass back null.
    // We could be passing in several different vectors for AllFVector
(AllF, AllFByCount,
    // AllFByProximity, etc)
    private Vector ScheduleOrders(FulfillerOrder fOrder, int schedulingMode)
  {

      Vector newFulfillerOrders = new Vector();
      Vector fOrderItems = fOrder.getItems();
      FulfillerOrderItem fOrderItem;
      Fulfiller fulfiller;
      ProductFulfiller productFulfiller;
      FulfillerOrder newFulfillerOrder;
      Product product;
      Vector AllFVector;
      Order order;
      OrderItem orderItem;
      int numberScheduled = 0;
      int i, j, k;

      for (i = 0; i < fOrderItems.size(); i++) {

          fOrderItem = (FulfillerOrderItem) fOrderItems.elementAt(i);

          order = fOrder.getParentOrder();
          orderItem = order.getOrderItem(fOrderItem.getOrderItemID());
          // we should use an orderItem but it is not set yet, use Order for
  now
          // Address address =
  Address.getAddressByID(orderItem.getShippingAddressID());
          //Address address =
  Address.getAddressByTypeAndID(Address.TYPE_SHIPPING,
  order.getShippingAddressID());
          Address address = null;
          try {
             address =
  service.selectMostRecentAddress(order.getShippingAddressID(),
  Address.TYPE_SHIPPING);
          } catch (Exception e) {
             System.err.println("ScheduleOrders: could not get
  selectMostRecntAddress");
             e.printStackTrace();
             return new Vector();
          }

          if (debug)
            System.out.println("zip is " + address.getPostalCode());
          if (schedulingMode == DistributeOrders.SCHEDULE_BY_PROXIMITY)
              AllFVector = byProximity(address.getPostalCode());
            else
              AllFVector = AllFByCount;

          for (j = 0; j < AllFVector.size(); j++) {

             // See if we can place the FulfillerOrderItem at this fulfiller
             fulfiller = (Fulfiller)AllFVector.elementAt(j);

             if (debug)
               System.out.println(fulfiller.getClass() + " fBV foiBV " +
  fulfiller.getBV() + " " + fOrderItem.getBV());

             if ((fulfiller.getBV() & fOrderItem.getBV()) ==
  fOrderItem.getBV()) {

               // place in HF, HP, VPi
               HP = (Hashtable) HF.get(new Integer(fulfiller.hashCode()));
               //order = fOrder.getParentOrder();
              //orderitem =
order.getOrderItem(fOrderItem.getFulfillerOrderItemID());
              VPi = (Vector) HP.get(new Long(orderItem.getProductID()));
              VPi.addElement(fOrderItem);
              fOrderItem.setFulfillerID(fulfiller.getFulfillerID());
              fOrder.setFulfillerID(fulfiller.getFulfillerID());
              numberScheduled++;
              break;
             }

        } // end for j
    } // end for i

    if (fOrderItems.size() == numberScheduled) {
        // This should always be the case, anything else is an error
        // Walk through HF, HP and VPi making a new FulfillerOrder for
each row of HF
        // that has any FulfillerOrderItems
        for (i = 0; i < fulfillers.size(); i++) {
            newFulfillerOrder = null;
            fulfiller = (Fulfiller)fulfillers.elementAt(i);
            HP = (Hashtable)HF.get(new Integer(fulfiller.hashCode()));
            for (j = 0; j < products.length; j++) {
                product = products[j];
                VPi = (Vector) HP.get(new Long(product.getProductID()));
                if (VPi.size() > 0) {
                   for (k = 0; k < VPi.size(); k++) {
                      if (newFulfillerOrder == null) {
                        newFulfillerOrder = new FulfillerOrder();

newFulfillerOrder.setFulfillerID(fOrder.getFulfillerID());
                        newFulfillerOrder.setOrderID(fOrder.getOrderID());

newFulfillerOrder.setParentOrder(fOrder.getParentOrder());

newFulfillerOrder.setStatus(FulfillerOrder.STATUS_PENDING);
                      }
                      newFulfillerOrder.addFulfillerOrderItem
                        ((FulfillerOrderItem)VPi.elementAt(k));
                   }
                }
            } // end for j, walk through the products

            if (newFulfillerOrder != null)
              newFulfillerOrders.addElement(newFulfillerOrder);
        } // end for i, walk through fulfillers
    } else {
      // This should be logged to a file !!! TBD
      try {
        if((Config.ERROR_LEVEL & Config.DEBUG_FULFILLMENT) > 0 )
          System.err.println("Error: fOrderItems != numScheduled for
fOrder=" +
          fOrder.getFulfillerOrderID());
      } catch (Exception e) {}
      System.out.println("Error!!! Orders scheduled different than number
of orderItems");
      return null;
    }

    // Clean up the scheduling data structure
    for (i = 0; i < fulfillers.size(); i++) {
        fulfiller = (Fulfiller)fulfillers.elementAt(i);
        HP = (Hashtable)HF.get(new Integer(fulfiller.hashCode()));
        for (j = 0; j < products.length; j++) {
                product = products[j];
                VPi = (Vector) HP.get(new Long(product.getProductID()));
                if (VPi.size() > 0) {
                    VPi.removeAllElements(); // thats it
                }
            } // end for j, walk through the products
        } // end for i, walk through fulfillers

        if (newFulfillerOrders.size() > 0)
          return newFulfillerOrders;
        else
          return null;

      } // end ScheduleOrders

      // Just updates the Order status field
      private void updateOrders(Vector newOrders, int status) {
        int i;
        Order order;
        if (newOrders != null) {
              for (i = 0; i < newOrders.size(); i++) {
                  try {
                  order = (Order)newOrders.elementAt(i);

                  order.setStatus(status);

                //order.update(dbc);
                service.updateOrder(order);
                } catch (Exception e) {
                    e.printStackTrace();
                }
              } // end for
          } // end for
      } // end updateOrders

      // Just updates the FulfillerOrder status field
      private void updateFulfillerOrders(Vector fulfillerOrders, int status) {
        int i;
        FulfillerOrder fOrder;
        if (fulfillerOrders != null) {
            DBContext  dbc = new DBContext();
            try {
                dbc.connect();
                  for (i = 0; i < fulfillerOrders.size(); i++) {
                    fOrder = (FulfillerOrder)fulfillerOrders.elementAt(i);
                      fOrder.setStatus(status);
                    fOrder.update(dbc);
                  }
            }
            catch(SQLException sqle) {}
            catch (Exception e) { e.printStackTrace(); }
            finally {
              try {
                    dbc.disconnect();
              }
              catch(SQLException sqle) {}
            }
          } // end for
      } // end updateFulfillerOrders

      // Return a list of fulfillers ordered by those closest to this zipCode
      public Vector byProximity (String zipCode) {
        int zoneOfZipCode;
        Vector vectorOfFulfillers = new Vector();
```

```
    Vector fulfillersTmp;
    int step = 1;
    boolean keepGoing;
    int i;

    // The first digit of a zip code is the "national area" of the
country.
    // The areas are:
    //   0 Northeast
    //   1 NewYork
    //   2 MidAtlantic
    //   3 Southeast
    //   4 GreatLakes
    //   5 Midwest
    //   6 Plains
    //   7 Southwest
    //   8 Western
    //   9 Pacific
    // This information is not online and I derived it by looking at post
office
    // maps. So the names may not be correct but it is close enough for
postal work.

    try {
        zoneOfZipCode = Integer.parseInt(zipCode.substring(0, 1));
    } catch(Exception e) {
        return vectorOfFulfillers; // passed in a malformed zip code
    }
    fulfillersTmp = Fulfiller.getByZone(zoneOfZipCode);
    for (i = 0; i < fulfillersTmp.size(); i++)
       vectorOfFulfillers.addElement(fulfillersTmp.elementAt(i));

    while (true) {
      keepGoing = false;

      // we may get a zone in the middle of the country so we need to step
      // away 1 zone at a time to make sure that we get the fulfillers
closest
      // to this zone

      if (zoneOfZipCode + step <= Fulfiller.LAST_ZIP_ZONE) {
        fulfillersTmp = Fulfiller.getByZone(zoneOfZipCode + step);
        for (i = 0; i < fulfillersTmp.size(); i++)
           vectorOfFulfillers.addElement(fulfillersTmp.elementAt(i));
        keepGoing = true;
      }

      if (zoneOfZipCode - step >= Fulfiller.FIRST_ZIP_ZONE) {
        fulfillersTmp = Fulfiller.getByZone(zoneOfZipCode - step);
        for (i = 0; i < fulfillersTmp.size(); i++)
           vectorOfFulfillers.addElement(fulfillersTmp.elementAt(i));
        keepGoing = true;
      }

      if (keepGoing == true)
        step++;
      else
        break;

    } // end while true
    return vectorOfFulfillers;
  }

}
```